Amendments to the Specification:

Please replace the paragraph beginning on page 1, line 29, with the following amended paragraph:

The ~~Java~~ JAVA® programming language simplifies the implementation of resource file-based solution by providing `java.util.ResourceBundle` class in the standard library. This class provides support for human language-specific resource files. However, if the programmer uses class `ResourceBundle` directly, the amount of code dedicated to localization still remains large. In addition, localization-specific code is typically spread all over the application code, making it difficult to read and maintain.

Please replace the paragraph beginning on page 2, line 5, with the following amended paragraph:

A method for localization of a ~~Java~~ JAVA® application includes locating a plurality of localizable variables of a class using a custom class loader, finding a corresponding resource file for a current language for each localizable variable, and calculating a key for each localizable variable. The method further includes finding a localized string in the resource file corresponding to each key, and assigning the localized string to the corresponding localizable variable of the class. Accordingly, the custom class loader provides localization of the class during class loading.

Please replace the paragraph beginning on page 2, line 12, with the following amended paragraph:

The method and corresponding apparatus for localization reduce complexity of the ~~Java~~ JAVA® application by eliminating the function code dedicated to localization. In addition, the method and apparatus for localization increase productivity of engineers who write language-independent code, and reduce memory consumption of classes that use localized strings. Since all localization is accomplished when the class is loaded and since the code that uses localization is independent of the code performing localization, the method and apparatus for localization ensure better performance of the application and afford better code reusability.

Please replace the paragraph beginning on page 3, line 16, with the following amended paragraph:

A method and corresponding apparatus for localization of a ~~Java~~ JAVA® application using a reflection API and a custom class loader use specifics of ~~Java~~ JAVA® language to provide localization of certain data elements, i.e., variables, of the application during class loading. The method and apparatus for localization provide a technique to localize static class variables, so that the data elements of a particular class are localized when the class loader loads that class. The method and apparatus typically apply to localization of user interface data elements defined as text.

Please replace the paragraph beginning on page 3, line 23, with the following amended paragraph:

The method and apparatus for localization reduce complexity of the ~~Java~~ JAVA® application by eliminating the function code dedicated to localization. The custom class loader is added to the application to support localization by inspecting the application code loaded from a storage device, such as a disk or Network, together with the application. The custom class loader may be added during code transferring process to inspect the code and convert all strings of the application to localized strings. As a result, the application may become less complicated, because many programmatic errors may be eliminated during the process.

Please replace the paragraph beginning on page 4, line 3, with the following amended paragraph:

Figure 1 illustrates a standard class loading process. Class files 115 are loaded from storage devices 110, such as disk or Network, to a bootstrap class loader 120. The bootstrap class loader 120 then converts the class files 115 into ~~Java~~ JAVA® classes 125 to be inputted in a ~~Java~~ JAVA® virtual machine 130. A class loader is an object that is responsible for loading classes. Given a name of a class, the class loader may attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name of the class into a file name and then read a "class file" of the name from the a file system. Applications typically implement subclasses of `ClassLoader` in order to extend the manner in which the ~~Java~~ JAVA® virtual machine 130 dynamically loads classes. The `ClassLoader` class uses a delegation model to search for classes and resources. Each instance of the `ClassLoader` class has an associated parent class loader. When asked to find a class or resource, the `ClassLoader` instance may delegate the search for the class or

resource to its parent class loader before attempting to find the class or resource itself. The ~~Java~~ JAVA® virtual machine's 130 built-in class loader, i.e., the bootstrap class loader 120, does not have a parent class loader, but may serve as the parent class loader of the `ClassLoader` instance. The ~~Java~~ JAVA® virtual machine 130 typically loads classes from a local file system in a platform-dependent manner. For example, on UNIX systems, the virtual machine loads classes from a directory defined by a `classpath` environment variable.

Please replace the paragraph beginning on page 4, line 22, with the following amended paragraph:

Figure 2 illustrates a custom class loading process that implements an exemplary method for localization using a reflection API and a custom class loader. Compared with the standard class loading process shown in Figure 1, a custom class loader 210 is developed during class loading to load the ~~Java~~ JAVA® classes 125 from the bootstrap class loader 120. In a ~~Java~~ JAVA® application, the custom class loader class 210 is typically a subclass of `java.lang.ClassLoader`. The custom class loader 210 also retrieves resource files 215 (described later) from the storage devices 110, and inspects the application code in the resource files 215. The custom class loader 210, together with a reflection API 230 and a launcher 220, then converts all strings of the application to localized strings. Finally, the custom class loader 210 passes localized ~~Java~~ JAVA® classes 225 to the ~~Java~~ JAVA® virtual machine 130. The reflection API 230 is a part of standard ~~Java~~ JAVA® library that obtains descriptors of class variables. The launcher 220 typically creates the custom class loader 210, loads a startup class 240 of an application using the custom class loader 210, and runs the application by invoking a main method of the application startup class 240. The application startup class 240 is a class that starts the application, whereas the main method is a method that initiates the application's functionality. The launcher 220 and the custom class loader 210 classes are loaded by the bootstrap class loader 120. The launcher 220 may need to use the reflection API 230 to invoke the main method of the application startup class 240. The launcher 220 may need to have a static method `main ()`, and be specified as a main class of the application, by, for example, including the launcher name as a parameter of a ~~Java~~ JAVA® command. The ~~Java~~ JAVA® virtual machine 130 may call the method `main ()` to invoke the main method of the application startup class 240.

Please replace the paragraph beginning on page 5, line 19, with the following amended paragraph:

Figure 3 is a flow chart illustrating an exemplary algorithm to be performed by a load class method of the custom class loader 210 to implement the exemplary method for localization. Referring to Figure 3, the custom class loader 210 first loads the resource files 215 from the storage device 110 (block 305), and locates all localizable variables of a class using the reflection API 230 and Rule 1 (block 310). A skilled ~~Java~~ JAVA® programmer may be able to implement this step given a definition of Rule 1. Then, for each localizable variable, the custom class loader 210 finds a corresponding localized resource file 215 for a current language according to Rule 2 by, for example, searching variables with predefined prefix or postfix with `LOC_` (block 320). All values in the resource files 215 may need to be specified before deploying the application.

Please replace the paragraph beginning on page 5, line 29, with the following amended paragraph:

Next, the custom class loader 210 calculates a key for each localizable variable according to Rule 2 by, for example, deleting the prefix and combining the class name and variable name (block 330). Then, the custom class loader 210 finds a localized string in the resource file 215 corresponding to each key (block 340). This step may be easily implemented by a skilled ~~Java~~ JAVA® programmer. Finally, the custom class loader 210 assigns the localized string to the corresponding localizable variable, if the string is found (block 350). If more localizable variables exist (block 360), blocks 320-350 repeat.

Please replace the paragraph beginning on page 6, line 29, with the following amended paragraph:

Figure 4 is a flow chart illustrating an exemplary algorithm to be performed by the launcher 220 (shown in Figure 2) to implement the exemplary method for localization. Referring to Figure 4, after the launcher 220 is created, the launcher 220 first creates an instance of the custom class loader 210 (block 410). Then, the launcher 220 requests and loads the application startup class 240 from the custom class loader 210, by calling a method, such as `Class.forName(String, ClassLoader)`, and specifies the application startup class name as a first parameter and the custom class loader instance as a second parameter (block 420). The custom class loader 210 localizes each class 240 loaded (block

~~442~~ 422), and the method returns a localized version of the application startup class 240 to the launcher 220. The custom class loader 210 may also load all classes referenced by the application startup class 240, since by default a class is loaded by the same class loader as the referencing class. For example, if class A refers to class B, and class B is not loaded yet, then the ~~Java~~ JAVA® virtual machine 130 uses the same class loader that was used for loading class A to load class B. The launcher 220 and the custom class loader 210 may not refer to any other application classes. Finally, the launcher 220 invokes the main method and runs the application startup class 240 using the reflection API 230 (block 430).

Please replace the paragraph beginning on page 7, line 13, with the following amended paragraph:

All classes of the application loaded by the custom class loader 210 may access the localizable variables, which may contain strings in the target human language at run time. As a result, the method and apparatus for localization allows a ~~Java~~ JAVA® application to use a single human language during execution. The target language can be specified either to the launcher 220 in the command line, or determined by the custom class loader 210 based on the default environment setting, or by any another method that is convenient for the implementation.

Please replace the paragraph beginning on page 7, line 27, with the following amended paragraph:

The resource file name for the target language is constructed by passing the base name and ~~Java~~ JAVA® Locale object to `java.util.ResourceBundle` class. The `ResourceBundle` class adds a postfix to the file name for locating the resource file 215 for specified human language. The resource file 215 is located in the same directory as the class file containing the variable. A key is constructed by concatenating the class name with variable name minus "`LOC_`" prefix, adding a dot (".") between the class name and variable name.

Please replace the paragraph beginning on page 8, line 23, with the following amended paragraph:

In this example, the implementation of the custom class loader 210 uses class `ResourceBundle` for retrieving localized strings by keys. The custom class loader 210

accepts the target language code as a parameter to its constructor, and passes the target language code to `ResoucreBundle`, which locates resource files 215 corresponding to the target language. The constructor is a method or procedure that is invoked when an object is created in order to initialize the variables of the object. A class may have more than one constructors, and the ~~Java~~ JAVA® virtual machine 130 has rules that define which constructor to call. Finally, the launcher 220 reads a configuration file containing the target language specified at the time of installation, and passes the code of target language to the custom class loader 210 to create the custom class loader 210. Alternatively, the custom class loader 210 may also read the configuration file. As a further alternative, the target language is passed as a parameter to the launcher 220 by a user.

Please replace the paragraph beginning on page 9, line 3, with the following amended paragraph:

Figure 5 illustrates exemplary hardware components of a computer 500 that may be used in connection with the method for ~~managing data from multiple data sources using conduits~~ localization using a reflection API and a custom class loader. The computer 500 includes a connection with a network 518 such as the Internet or other type of computer or telephone networks. The computer 500 typically includes a memory 502, a secondary storage device 512, a processor 514, an input device 516, a display device 510, and an output device 508.

Please replace the abstract with the following amended Abstract:

ABSTRACT

A method and corresponding apparatus for localization of a ~~Java~~ JAVA® application using a reflection API and a custom class loader use specifics of ~~Java~~ JAVA® language to provide localization of certain data elements, i.e., variables, of the application during class loading. The method and corresponding apparatus for localization reduce complexity of the ~~Java~~ JAVA® application by eliminating the function code dedicated to localization. In addition, the method and apparatus for localization increase productivity of engineers who write language-independent code, and reduce memory consumption of classes that use localized strings. Since all localization is accomplished when the class is loaded and since the code that uses localization is independent of the code performing localization, the method

and apparatus for localization ensure better performance of the application and afford better code reusability.